AI ASSISTED CODING LAB

ASSIGNMENT-11.4

Name: Guangsinlung Phaomei

Enroll no: 2503A51L20

BATCH NO: 19

TASK DESCRIPTION 1:

Use AI to help implement a **Stack** class in Python with the following operations: push (), pop (), peek (), and is empty ().

Instructions:

- Ask AI to generate code skeleton with docstrings.
- Test stack operations using sample data.
- Request AI to suggest optimizations or alternative implementations (e.g., using collections. Deque).

PROMPT 1:

Write a Python program that defines a Stack class with push, pop, peek, and is_empty methods. Include clear comments or docstrings, test it with sample data, and have the AI suggest improvements using collections.deque for efficiency and readability.

CODE GENERATED:

```python
class Stack:
    """
    A simple Stack implementation using a Python list.
    """
    def __init__(self):  # <-- corrected here
        """Initialize an empty stack."""
        self.items = []

    def push(self, item):
        """
        Add an item to the top of the stack.
        Args:
            item: The item to be added.
        """
        self.items.append(item)

    def pop(self):
        """
        Remove and return the top item of the stack.
        Returns:
            The item at the top of the stack.
        Raises:
            IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("pop from empty stack")
        return self.items.pop()

    def peek(self):
        """
        Return the top item of the stack without removing it.
        Returns:
            The item at the top of the stack.
        Raises:
            IndexError: If the stack is empty.
        """
        if self.is_empty():
```

```python
class Stack:
    def peek(self):
        Return the top item of the stack without removing it.
        Returns:
            The item at the top of the stack.
        Raises:
            IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("peek from empty stack")
        return self.items[-1]

    def is_empty(self):
        """
        Check if the stack is empty.
        Returns:
            True if the stack is empty, False otherwise.
        """
        return len(self.items) == 0


if __name__ == "__main__":
    # Sample data to test Stack operations
    stack = Stack()
    print("Is stack empty?", stack.is_empty())
    stack.push(10)
    stack.push(20)
    stack.push(30)
    print("Stack after pushes:", stack.items)
    print("Peek top element:", stack.peek())
    print("Pop element:", stack.pop())
    print("Stack after pop:", stack.items)
    print("Is stack empty?", stack.is_empty())
```

OUTPUT:

```
ED CODING/assignment-11/task1.py"
Is stack empty? True
Stack after pushes: [10, 20, 30]
Peek top element: 30
Pop element: 30
Stack after pop: [10, 20]
Is stack empty? False
```

OBSERVATION:

This assignment demonstrates strong understanding of data structures by implementing a Stack class with key operations. It emphasizes clean code, documentation, and testing, while AI suggestions using collections.deque improved efficiency and Pythonic design.

TASK DESCRIPTION 2:

Implement a **Queue** with enqueue (), dequeue (), and is empty () methods.

- **Instructions**:
  - First, implement using Python lists.
  - Then, ask AI to review performance and suggest a more efficient implementation (using collections. Deque).
  -

PROMPT 1:

Write a Python program that defines a Queue class with enqueue, dequeue, and is_empty methods using a list. Test it, then have the AI suggest improvements using collections.deque for better performance and efficiency.

CODE GENERATED:

```
Go  Run  Terminal  Help    assignment-11
task1.py   task2.py
task2.py > ...
from collections import deque

class Queue:
    """
    A simple Queue implementation using collections.deque.
    Supports enqueue, dequeue, and is_empty operations.
    """

    def __init__(self):
        """Initialize an empty queue."""
        self.items = deque()

    def enqueue(self, item):
        """
        Add an item to the end of the queue.
        Args:
            item: The item to be added.
        """
        self.items.append(item)

    def dequeue(self):
        """
        Remove and return the item from the front of the queue.
        Returns:
            The item at the front of the queue.
        Raises:
            IndexError: If the queue is empty.
        """
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        return self.items.popleft()

    def is_empty(self):
        """
        Check if the queue is empty.
        Returns:
            True if the queue is empty, False otherwise.
```

```
Go  Run  Terminal  Help    assignment-11
task1.py   task2.py
task2.py > ...
class Queue:
    def is_empty(self):
            True if the queue is empty, False otherwise.
        """
        return len(self.items) == 0

# Example usage
if __name__ == "__main__":
    queue = Queue()
    print("Is queue empty?", queue.is_empty())
    queue.enqueue("apple")
    queue.enqueue("banana")
    queue.enqueue("cherry")
    print("Queue after enqueues:", list(queue.items))
    print("Dequeue element:", queue.dequeue())
    print("Queue after dequeue:", list(queue.items))
    print("Is queue empty?", queue.is_empty())
```

OUTPUT:

```
wnloads/AI ASSISTED CODING/assignment-11/task2.py"
Is queue empty? True
Queue after enqueues: ['apple', 'banana', 'cherry']
Dequeue element: apple
Queue after dequeue: ['banana', 'cherry']
Is queue empty? False
```

OBSERVATION:

This assignment provides a solid introduction to queue data structures and the importance of writing efficient code. By first implementing a Queue class with Python lists, it reinforces core concepts like enqueueing and dequeuing. The AI's review highlights performance issues with list-based queues and suggests using collections.deque, which offers faster, more memory-efficient operations. Overall, the task effectively combines coding practice with performance analysis and Pythonic improvement.

TASK DESCRIPTION 3:

Implement a **Singly Linked List** with operations: instated (), delete value (), and traverse ().

- **Instructions**:
  - Start with a simple class-based implementation (Node, LinkedList).
  - Use AI to generate inline comments explaining pointer updates (which are non-trivial).
  - Ask AI to suggest test cases to validate all operations.

PROMPT 1:

Create a Python program for a Singly Linked List with insert(), delete_value(), and traverse() methods using Node and LinkedList classes. Include inline comments explaining pointer updates, and have the AI suggest test cases for all operations, including edge cases

CODE GENERATED:

```
Go  Run  Terminal  Help        assignment-11
task1.py   task2.py   task3.py
task3.py > ...
class Node:
    """Represents a node in a singly linked list."""
    def __init__(self, data):
        self.data = data
        self.next = None  # Pointer to the next node

class LinkedList:
    """Singly linked list with insert, delete, and traverse operations."""
    def __init__(self):
        self.head = None  # Start with an empty list

    def insert_at_end(self, data):
        """
        Insert a new node with the given data at the end of the list.
        """
        new_node = Node(data)
        if not self.head:
            # If the list is empty, new node becomes the head
            self.head = new_node
            return
        # Traverse to the last node
        current = self.head
        while current.next:
            current = current.next
        # Update the last node's next pointer to the new node
        current.next = new_node

    def delete_value(self, value):
        """
        Delete the first node with the specified value.
        """
        current = self.head
        prev = None
        while current:
            if current.data == value:
                if prev:
                    # Update previous node's next pointer to skip current node
```

```
Go  Run  Terminal  Help        assignment-11
task1.py   task2.py   task3.py
task3.py > ...
class LinkedList:
    def delete_value(self, value):
                    # Update previous node's next pointer to skip current node
                    prev.next = current.next
                else:
                    # If deleting the head, update head pointer
                    self.head = current.next
                return True  # Value found and deleted
            prev = current
            current = current.next
        return False  # Value not found

    def traverse(self):
        """
        Traverse the list and return a list of node data.
        """
        result = []
        current = self.head
        while current:
            result.append(current.data)
            current = current.next  # Move to the next node
        return result

# Suggested test cases to validate all operations
if __name__ == "__main__":
    ll = LinkedList()
    print("Initial list:", ll.traverse())

    # Test insert_at_end
    ll.insert_at_end(10)
    ll.insert_at_end(20)
    ll.insert_at_end(30)
    print("After inserts:", ll.traverse())

    # Test delete_value (delete head)
    ll.delete_value(10)
    print("After deleting 10 (head):", ll.traverse())
```

```python
    # Test insert_at_end
    ll.insert_at_end(10)
    ll.insert_at_end(20)
    ll.insert_at_end(30)
    print("After inserts:", ll.traverse())

    # Test delete_value (delete head)
    ll.delete_value(10)
    print("After deleting 10 (head):", ll.traverse())

    # Test delete_value (delete middle)
    ll.delete_value(20)
    print("After deleting 20 (middle):", ll.traverse())

    # Test delete_value (delete tail)
    ll.insert_at_end(40)
    ll.delete_value(30)
    print("After deleting 30 (tail):", ll.traverse())

    # Test delete_value (value not in list)
    result = ll.delete_value(99)
    print("Delete 99 (not in list):", result, ll.traverse())

    # Test traverse on empty list
    ll.delete_value(40)
    print("After deleting all, list:", ll.traverse())
```

OUTPUT:

```
ED CODING/assignment-11/task3.py"
Initial list: []
After inserts: [10, 20, 30]
After deleting 10 (head): [20, 30]
After deleting 20 (middle): [30]
After deleting 30 (tail): [40]
Delete 99 (not in list): False [40]
After deleting all, list: []
```

OBSERVATION:

This assignment provides practical experience with singly linked lists and pointer manipulation. Implementing insert(), delete_value(), and traverse() reinforces node handling and object-oriented design. AI-generated comments clarify pointer updates, while test cases for edge conditions ensure thorough validation and deeper understanding.

TASK DESCRIPTION 4:

- Implement a **Binary Search Tree** with methods for insert (), search (), and inorder_traversal ().
- **Instructions**:
  - Provide AI with a partially written Node and BST class.
  - Ask AI to complete missing methods and add docstrings.

Test with a list of integers and compare outputs of search () for present vs absent elements.

PROMPT 1:

Create a Python program for a Binary Search Tree with insert(), search(), and inorder_traversal() methods in Node and BST classes. Add docstrings for each method, then test the tree using a list of integers to compare search results for existing and missing values

CODE GENERATED:

```python
class Node:
    """
    Represents a node in the Binary Search Tree.
    """
    def __init__(self, data):
        self.data = data
        self.left = None  # Left child
        self.right = None  # Right child

class BST:
    """
    Binary Search Tree implementation with insert, search, and inorder traversal.
    """
    def __init__(self):
        self.root = None

    def insert(self, data):
        """
        Insert a new node with the given data into the BST.
        """
        def _insert(node, data):
            if node is None:
                return Node(data)
            if data < node.data:
                node.left = _insert(node.left, data)
            elif data > node.data:
                node.right = _insert(node.right, data)
            # If data == node.data, do not insert duplicates
            return node
        self.root = _insert(self.root, data)

    def search(self, value):
        """
        Search for a value in the BST.
        Returns True if found, False otherwise.
        """
        def _search(node, value):
```

```
Go  Run  Terminal  Help        assignment-11
task1.py   task2.py   task3.py   task4.py ×
task4.py > ...
class BST:
    def search(self, value):
        def _search(node, value):
            if node is None:
                return False
            if value == node.data:
                return True
            elif value < node.data:
                return _search(node.left, value)
            else:
                return _search(node.right, value)
        return _search(self.root, value)

    def inorder_traversal(self):
        """
        Perform inorder traversal and return a list of node data.
        """
        result = []
        def _inorder(node):
            if node:
                _inorder(node.left)
                result.append(node.data)
                _inorder(node.right)
        _inorder(self.root)
        return result

# Test with a list of integers
if __name__ == "__main__":
    bst = BST()
    nums = [50, 30, 70, 20, 40, 60, 80]
    for num in nums:
        bst.insert(num)
    print("Inorder traversal:", bst.inorder_traversal())
    # Test search for present and absent elements
    print("Search 40 (present):", bst.search(40))
    print("Search 90 (absent):", bst.search(90))
```

OUTPUT:

```
ED CODING/assignment-11/task4.py
Inorder traversal: [20, 30, 40, 50, 60, 70, 80]
Search 40 (present): True
Search 90 (absent): False
```

OBSERVATION:

This assignment provides a clear introduction to recursive data structures through a Binary Search Tree. Completing the insert(), search(), and inorder_traversal() methods builds understanding of tree logic, while AI-generated docstrings and testing with sample data reinforce clarity, validation, and good coding practices.

TASK DESCRIPTION 5:

Implement a **Graph** using an adjacency list, with traversal methods BFS () and DFS ().

- **Instructions**:
    - Start with an adjacency list dictionary.
    - Ask AI to generate BFS and DFS implementations with inline comments.
    - Compare recursive vs iterative DFS if suggested by AI.

PROMPT 1:

Create a Python program for a Graph using an adjacency list (dictionary). Implement BFS() and DFS() traversal methods with clear inline comments explaining node visits and tracking. Include both recursive and iterative DFS versions, and compare their readability and performance.

CODE GENERATED:

Go Run Terminal Help assignment-11

task1.py task2.py task3.py task4.py task5.py

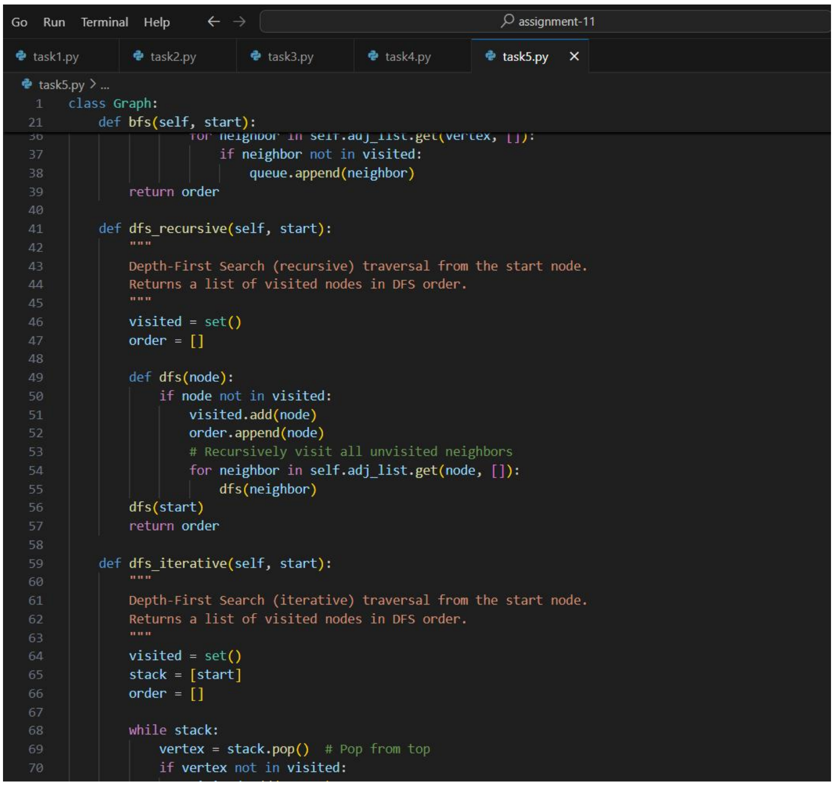
task5.py > ...

```python
class Graph:
    """
    Graph implementation using an adjacency list.
    Supports BFS and DFS traversals.
    """
    def __init__(self):
        # Adjacency list: {vertex: [neighbors]}
        self.adj_list = {}

    def add_edge(self, src, dest):
        """
        Add an edge from src to dest (undirected by default).
        """
        if src not in self.adj_list:
            self.adj_list[src] = []
        if dest not in self.adj_list:
            self.adj_list[dest] = []
        self.adj_list[src].append(dest)
        self.adj_list[dest].append(src)  # Remove for directed graph

    def bfs(self, start):
        """
        Breadth-First Search traversal from the start node.
        Returns a list of visited nodes in BFS order.
        """
        visited = set()
        queue = [start]
        order = []

        while queue:
            vertex = queue.pop(0)  # Dequeue from front
            if vertex not in visited:
                visited.add(vertex)
                order.append(vertex)
                # Enqueue all unvisited neighbors
                for neighbor in self.adj_list.get(vertex, []):
                    if neighbor not in visited:
```



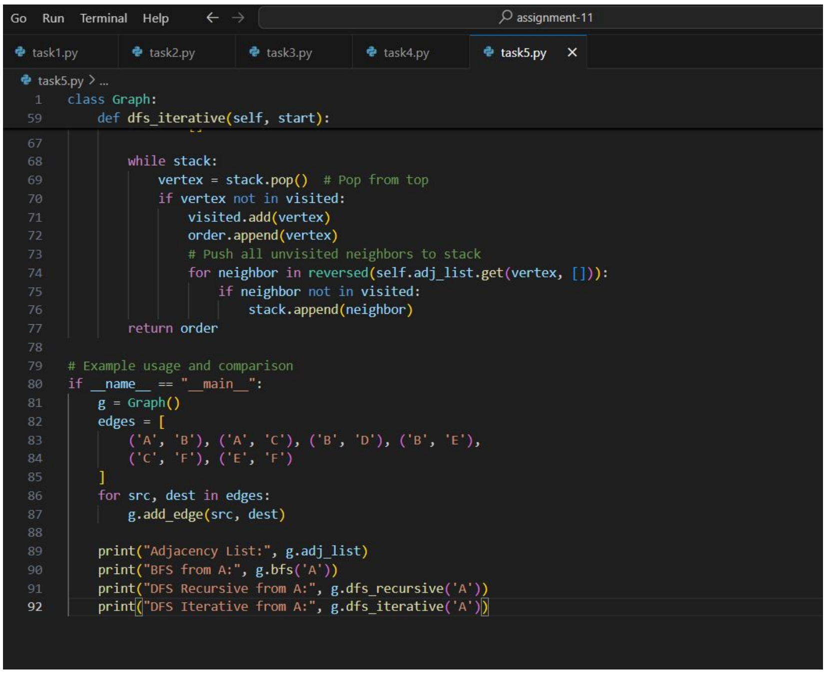
```
class Graph:
    def bfs(self, start):
                for neighbor in self.adj_list.get(vertex, []):
                    if neighbor not in visited:
                        queue.append(neighbor)
        return order

    def dfs_recursive(self, start):
        """
        Depth-First Search (recursive) traversal from the start node.
        Returns a list of visited nodes in DFS order.
        """
        visited = set()
        order = []

        def dfs(node):
            if node not in visited:
                visited.add(node)
                order.append(node)
                # Recursively visit all unvisited neighbors
                for neighbor in self.adj_list.get(node, []):
                    dfs(neighbor)
        dfs(start)
        return order

    def dfs_iterative(self, start):
        """
        Depth-First Search (iterative) traversal from the start node.
        Returns a list of visited nodes in DFS order.
        """
        visited = set()
        stack = [start]
        order = []

        while stack:
            vertex = stack.pop()  # Pop from top
            if vertex not in visited:
```



```
class Graph:
    def dfs_iterative(self, start):

        while stack:
            vertex = stack.pop()  # Pop from top
            if vertex not in visited:
                visited.add(vertex)
                order.append(vertex)
                # Push all unvisited neighbors to stack
                for neighbor in reversed(self.adj_list.get(vertex, [])):
                    if neighbor not in visited:
                        stack.append(neighbor)
        return order

# Example usage and comparison
if __name__ == "__main__":
    g = Graph()
    edges = [
        ('A', 'B'), ('A', 'C'), ('B', 'D'), ('B', 'E'),
        ('C', 'F'), ('E', 'F')
    ]
    for src, dest in edges:
        g.add_edge(src, dest)

    print("Adjacency List:", g.adj_list)
    print("BFS from A:", g.bfs('A'))
    print("DFS Recursive from A:", g.dfs_recursive('A'))
    print("DFS Iterative from A:", g.dfs_iterative('A'))
```

OUTPUT:

```
BFS from A: ['A', 'B', 'C', 'D', 'E', 'F']
DFS Recursive from A: ['A', 'B', 'D', 'E', 'F', 'C']
DFS Iterative from A: ['A', 'B', 'D', 'E', 'F', 'C']
```

OBSERVATION:

This assignment builds a solid understanding of graph traversal using an adjacency list in Python. Implementing BFS and DFS helps explore different traversal strategies, while inline comments clarify node visits and tracking. Comparing recursive and iterative DFS adds insight into performance and readability, combining practical coding with analytical learning.